VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Generátor cílového kódu pro Vestavný procesně funkcionální jazyk

Target code generator for Embedded process functional language

2010 Václav Kufa

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 1. května 2010 .....

Abstrakt

Programovací jazyk e-PFL (Embedded Process Functional Language, Vestavný procesně funkcionální jazyk) je určen pro modelování vestavných systémů na vysoké úrovni abstrakce. Cílem mé bakalářské práce bylo rozšírit již stávající kompilátor o možnost překladu zdrojového kódu v jazyce e-PFL do programovacího jazyka Erlang. Erlang je jazyk integrující prvky funkcionálního paradigmatu a je hojně využíván pro komerční účely, zejména v telekomunikacích. Při realizaci bylo nejdřív nutné vyjádřit specifické konstrukce jazyka e-PFL v jazyce Erlang tak, aby funkčnost modelovaného vestavného systému byla stejná v obou jazycích. Syntaxe i sémantika obou jazyků jsou podobné, ale jinak se jazyky v mnoha ohledech liší. Jazyk e-PFL nabízí mechanismus pro snadnější modelování vestavných systémů. Bylo proto nutné navrhnout, jak převést konstrukce z jazyka e-PFL do jazyka Erlang. Výsledný generátor cílového kódu byl implementován do existujícího překladače, ten je napsán v jazyce C#.

Klíčová slova: Erlang, e-PFL, kompilátor, funkcionální jazyky, vestavné systémy

Abstract

The e-PFL (Embedded Process Functional Language) programming language is made for a high level abstraction modeling of embedded systems. Goal of my bachelor thesis was to extend already existing compiler, so it would translate a source code written in the e-PFL programming language to the source code in Erlang programming language. Erlang is a programming language integrating the characteristics of a functional paradigm and is widely used for a commercial purposes, mainly in telecommunications. First important thing was to express special constructions of the e-PFL in the Erlang, thus the functionality of a modeled embedded system would be equal in both of these languages. Syntax and semantics of both languages are similar, otherwise they have many differences. In e-PFL programming language, the mechanism for easier embedded system modeling is provided. Thus it was necessary to design the way, of the translation the language constructions from e-PFL to Erlang. Final target code generator was implemented into the existing compiler and it is written in the C# programming language.

Keywords: Erlang, e-PFL, compiler, functional languages, embedded systems

Obsah

1 Úvod **3**
1.1 Obsah práce 3

2 Funkcionální programování **4**
2.1 Typování ve funkcionálních programovacích jazycích 4
2.2 Funkce vyššího řádu 5
2.3 Rekurze 5
2.4 Pattern matching 5

3 Programovací jazyky používané pro vestavné systémy **7**

4 Erlang **8**
4.1 Požadavky kladené na telekomunikační systémy 8
4.2 Hlavní cíle při vývoji programovacího jazyka Erlang 8
4.3 Klíčové vlastnosti jazyka Erlang 9
4.4 OTP (Open Telecom Platform) 12
4.5 Erlang v reálných aplikacích 12

5 Programovací jazyk e-PFL **13**
5.1 PFL 13
5.2 Koordinační vrstva jazyka e-PFL 14
5.3 Komunikace funkčních jednotek v e-PFL 14
5.4 Konfigurace systému v e-PFL 15

6 Praktická realizace rozšíření kompilátoru **17**
6.1 Datový typ Device 17
6.2 Implementace datového typu Device v programovacím jazyce Erlang . . . 18
6.3 Rozšíření kompilátoru 26
6.4 Aktuální stav implementace 29

7 Ukázka vygenerovaného cílového kódu **30**

8 Závěr **34**

9 Reference **35**

Přílohy **35**

A Zdrojové kódy **36**

B Obsah CD **38**

Seznam výpisů zdrojového kódu

1 Ukázka zásobníku (*stack*) . . . 9
2 Quicksort v jazyce Erlang . . . 10
3 Ukázka procesu v jazyce Erlang . . . 10
4 Ukázka *hot code loading* . . . 11
5 Ukázka zdrojového kódu v jazyce e-PFL . . . 15
6 Ukázka konfigurace systému . . . 16
7 Implementace stromu . . . 20
8 Funkce shuffle . . . 21
9 Funkce output_creator . . . 22
10 Spouštění zařízení . . . 22
11 Spuštění systému . . . 23
12 Funkce sw_receiver . . . 24
13 Funkce switch . . . 24
14 Funkce input_box . . . 25
15 Funkce tree_holder . . . 25
16 Funkce process_caller . . . 26
17 Funkce v jazyce e-PFL obsahující lokální výraz . . . 29
18 Funkce v jazyce Erlang obsahující lokální výraz . . . 29
19 Kód v jazyce e-PFL, 1. část . . . 30
20 Cílový kód v jazyce Erlang, 1. část . . . 30
21 Cílový kód v jazyce Erlang, pomocné funkce . . . 30
22 Kód v jazyce e-PFL, 2. část . . . 31
23 Cílový kód v jazyce Erlang, 2. část . . . 31
24 Kód v jazyce e-PFL, 3. část . . . 33
25 Cílový kód v jazyce Erlang, 3. část . . . 33
26 Průchod konfigurací . . . 36
27 Funkce search . . . 37

1 Úvod

Vestavné systémy v dnešní době tvoří významnou část informatiky. Bez přílišné nadsázky se dá říci, že vestavné systémy jsou všude. Ovládají kuchyňské spotřebiče, drobnou elektroniku nebo třeba automobily - od motoru až po elektronické sklápění zpětných zrcátek. Vestavným systémem je možno rozumět veškeré elektronické systémy kromě osobních a sálových počítačů. Tomu také odpovídají prodeje těchto zařízení: na jeden prodaný osobní počítač připadá asi 1000 vestavných systémů.
Vestavné systémy mají společné vlastnosti, díky čemuž také můžeme klasifikovat, co vestavný systém je a co ne. První vlastností je omezená funkčnost, schopnost provádět pouze jeden či několik úkonů. Například digitální budík umí pouze zazvonit v určitou, předem nastavenou dobu. Důležitou vlastností vestavných systémů je také práce v reálném čase. V případě elektronických bezpečnostních systémů v automobilech je tato vlastnost přímo kritická. Dalším jejich rysem (jako i všech počítačů) jsou jistě jejich omezené zdroje - třeba nevýkonný procesor, malá paměť nebo napájení z baterie. Návrh jakéhokoliv vestavného systému se skládá z návrhu hardwaru a návrhu softwaru. Tyto dva pohledy na vestavný elektronický systém spolu také velmi blízce souvisí. Pro implemetaci softwaru se jako velmi vhodné jeví funkcionální programovací paradigma.

1.1 Obsah práce

Funkcionální programování je jedním z programovacích přístupů. Jeho stručnému popisu je věnovaná kapitola 2. Mimo jiné ho lze s úspěchem použít právě ve vestavných systémech (technologie používané pro implementaci vestavných systémů jsou zmíněny v kapitole 3). Jedním z funkcionálních jazyků je také Erlang. Na rozdíl od jiných funkcionálních jazyků, které mívají původ v akademickém prostředí, Erlang vznikl pro komerční využití, zejména v telekomunikacích. Byl tak již od začátku vyvíjen s jasnými požadavky, díky kterým obsahuje užitečné, ba dokonce nezbytné vlastnosti. Ty jsou pro návrh vestavných systémů velmi důležité (vždyt'i různé komponenty telekomunikačních sítí jsou vlastně vestavnými systémy), a proto je možné Erlang v této oblasti informatiky použít. Programovacímu jazyku Erlang je věnována kapitola číslo 4, v níž se především zaměřuji na vlastnosti, které jsou pro něj typické. Cílem této bakalářské práce bylo rozšíření funkčnosti již existujícího kompilátoru o možnost překladu do jazyka Erlang. Tento kompilátor v současné době překládá procesně funcionální jazyk *e-PFL* (*Embedded Process Functional Language*) do funkcionálního jazyka Hume, a do jazyka C# na platformě .NET Micro. Jazyk e-PFL byl speciálně vytvořen pro nasazení do vestavných systémů. Popis a vlastnosti procesně funkcionálního jazyka e-PFL jsou uvedeny v kapitole 5.
V části 6 je uveden průběh vlastní implementace rozšíření kompilátoru: příprava, problémy, s nimiž jsem se setkal, a popis části kompilátoru, která zodpovídá za generování cílového kódu pro vestavné systémy v jazyce Erlang. V kapitole číslo 7 je potom srovnání vygenerovaného cílového kódu v jazyce Erlang s původním kódem v jazyce e-PFL.

2 Funkcionální programování

Funkcionální paradigma [1, 5] vychází z teoretického modelu uplatňujícího se také v matematické logice. Nazývá se λ *kalkul* (λ-*calculus*). Ten vznikl ve třicátých letech minulého století a jeho tvůrcem byl americký matematik Alonso Church. λ kalkul je formální systém, který určuje definici funkce, její realizaci a rekurzi. Prvním funkcionálním jazykem (a druhým nejstarším programovacím jazykem vůbec) byl LISP, jež byl λ kalkulem přímo inspirován. Existuje také čistě funkcionální verze LISPu, Pure LISP, která je ekvivalentní λ kalkulu.
V současné době existuje množství programovacích paradigmatů. Pravděpodobně nejznámější a nejpoužívanější je programování *imperativní* (*imperative programming*) a *deklarativní* (*declarative programming*). V imperativním přístupu se stav programu mění na základě různých příkazů (nejjednodušším z nich může být například přiřazení hodnoty k proměnné: x = 1). Imperativní programování se v tom smyslu podobá rozkazovacímu způsobu lidské řeči (jak už napovídá samotný název), programátor definuje sérii příkazů, které má počítač provést. Jinak řečeno, udává, CO a JAK se má vykonat. V dnešní době je nejrozšířenější způsob programování *objektově orientované programování* (*object-oriented programming*). Jazyky jako C, C++, C# nebo Java, jež jsou jeho hlavními zástupci, jsou zároveň také zástupci imperativního přístupu.
Naproti tomu v deklarativním přístupu (pod něž se řadí také funkcionální paradigma) programátor neudává posloupnost příkazů, které by říkaly, JAK (nepopisuje se zde *control flow* programu) se co má vykonat, udává pouze, CO se má vykonat.
Fukcionální program se skládá z funkcí, lépe řečeno z čistě funkcionálních funkcí (*pure functions*), což jsou funkce postrádající vedlejší efekty (*side effects*). Každá funkce má daný počet argumentů (přičemž nemusí mít žádný) a produkuje určitý výstup, výsledek. Argumentem funkce může být jiná funkce. Výsledkem funkce může být také jiná funkce. Běh programu tak znamená vykonávání posloupnosti funkcí. Funkcionální program postrádá *stav* a *proměnné* v pravém slova smyslu - proměnným, kterým byla již jednou přiřazena hodnota, nemůže být přiřazena hodnota jiná. Toto je další rozdíl oproti imperativnímu programování, pro něhož jsou stav a změna hodnoty proměnné důležitými prvky. Ve funkcionálních jazycích lze, co se vyhodnocování výrazů (*evaluation strategy*) týká, narazit jak na vyhodnocování na základě hodnoty (*Eager evaluation*), tak na vyhodnocování na základě požadavku (*Lazy evaluation*). Obecně lze říci, že vlastnosti jednoho programovacího paradigmatu jsou cizí paradigmatu druhému, nemusí to ale platit vždy. Mezi výhody funkcionálního programování patří jeho některé, dále uvedené, specifické vlastnosti.

2.1 Typování ve funkcionálních programovacích jazycích

Typování (*typing*), neboli přiřazování datového typu k proměnné, určuje, jak má být hodnota v počítači chápána. Jelikož může být nějaké číslo v počítači vyjádřeno stejnou bitovou posloupností, jako třeba nějaké písmeno nebo instrukce, je samozřejmě typování velmi důležité. Podle toho, kdy je na proměnnou uplatněn daný datový typ a kdy proběhne kontrola typů (například jestli jsou oba sčítance čísly a ne booleanovskými hodnotami) -

proces označovaný jako *type checking* - rozlišujeme statickou kontrolu (*static check*) a dynamickou kontrolu (*dynamic check*). Statická kontrola probíhá při kompilaci. Dynamická kontrola naopak probíhá při běhu programu.
To také určuje druh typování určitého programovacího jazyka - statické typování (*static typing*) a dynamické typování (*dynamic typing*). Statické typování může kontrolovat pouze hodnoty, které jsou známé v době kompilace, proto může hlásit chyby tam, kde by při běhu programu nevznikly. Na druhé straně je schopen odhalit chyby ve všech částech kódu. Mezi staticky typované funkcionální programovací jazyky patří Haskell nebo ML. U dynamického typování mají datový typ všechny hodnoty, na rozdíl od jejich identifikátorů, které ho nemají (identifikátor může mít hodnotu jakéhokoliv typu). Dynamicky typovaný program, který prošel kompilací, může skončit chybou při běhu, je-li někde použita nevyhovující hodnota. Právě Erlang je dynamicky typovaným jazykem. Zjednodušeně řečeno, píšeme-li kód v staticky typovaném jazyce, u každé proměnné uvádíme její datový typ, kdežto u dynamicky typovaných jazyků jsou datové typy odvozeny kompilátorem, to znamená, že jsou známy až v době kompilace. Jazyk e-PFL je zároveň staticky i dynamicky typovaným jazykem.

2.2 Funkce vyššího řádu

Funkce vyššího řádu jsou definovány jako funkce, které operují nad jinými funkcemi, jinak řečeno to jsou takové funkce, které mohou mít jinou funkci/funkce jako své argumenty nebo mohou nějakou funkci/funkce vracet jako výsledek. Funkce vyššího řádu jsou podobné funkcím první třídy, které mohou být použity všude tam, kde jiné prvky první třídy (což jsou třeba čísla nebo booleanovské hodnoty), tzn. jako argumenty a výsledky funkcí. Funkce vyššího řádu umožňují *currying* - proces, který přemění funkci s n argumenty na n funkcí, každou pouze s jedním argumentem.

2.3 Rekurze

Iterace se ve funkcionálních jazycích řeší pomocí rekurze. Jazyk Erlang vůbec neobsahuje iterační konstrukce dobře známé z jiných jazyků jako *for* nebo *while*. Jde o to, že funkci voláme znova v jejím těle (funkce volá sebe samu), ale (většinou) s jinými parametry, čímž můžeme simulovat třeba cyklus *for* - funkci vždy předáme inkrementovaný/dekrementovaný čítač. Pokud je poslední výraz v těle funkce rekurzivní volání, označujeme rekurzi jako *koncovou* (*tail recursion*). Při koncové rekurzi nejsou spotřebovány systémové zdroje. Lze tak zrealizovat nekonečnou smyčku.

2.4 Pattern matching

Pattern matching je mechanismus používaný ve funkcionálních jazycích. Na základě splnění podmínky vykoná příslušnou klauzuli (jako například řídící struktura *if-else if-else* používaná v objektově orientovaných jazycích). Jako podmínka zde slouží "šablona" (*pattern*), která představuje datový typ nebo přímo hodnotu, pro níž má daná klauzule platit. Tzn. klauzule je splněna, když se šablona shoduje s tím, co je s ní porovnáváno. Výhodou

je poměrně velká abstrakce při navrhování šablony. Na jedné straně se dá udělat tak, že jí projde všechno, na straně druhé jí však může vyhovovat pouze přesně požadovaný vstup. Při tvorbě šablon (například při zpracování seznamů) je potřeba si dávat pozor, aby se klauzule svými požadavky nepřekrývaly, mohla by tak být splněna jiná klauzule, než jakou jsme chtěli. Vedle definice jednotlivých šablon totiž hraje roli i jejich pořadí - vykonána je první klauzule, jejíž šablona se shoduje se vstupem, další už nejsou porovnávány. V jazyce Erlang je možno využít pattern matching jak při funkčních volání, tak při přijímání zpráv od jiných procesů. V následujícím výpisu zdrojového kódu je ukázka pattern matchingu v jazyce Erlang. První funkce, *list_search*, slouží k nalezení hodnoty *true* v seznamu. Pokud aktuální seznam splňuje podmínku a na první místě je *true*, funkce končí. Pokud je na prvním místě hodnota *false*, funkce rekurzivně pokračuje na zbytku seznamu (vyjádřeno proměnnou *T* - může to být seznam s deseti prvky, stejně tak jako prázdný seznam). Prvek nebyl nalezen až v případě, kdy se funkce dostane k prázdnému listu (první klauzule). Ve funkci *tuple_check* jsou klauzule splněny vždy v pouze jednom konkrétním případě - když je přijatá dvojice shodná s tou, uvedenou v šabloně.

3 Programovací jazyky používané pro vestavné systémy

Neexistuje programovací paradigma, které by bylo pro použití ke tvorbě vestavných systémů výrazně lepší než paradigmata jiná. Funkcionální přístup má své výhody i nevýhody. Nevýhodami jsou ztráta části čistého výkonu (což v dnešní není moc velký problém vzhledem k cenám a výkonnostem procesorů, navíc tyto ztráty se dají zlepšit optimalizací) a ztráta přesné kontroly nad prováděnými operacemi. Naopak však nabízí mnohem větší míru abstrakce, dobře použitelnou pro modelování vestavného systému, než ostatní programovací přístupy. Také stručnost funkcionálního kódu je výhodou - je méně náchylný k chybám a je snadnější na údržbu.
Vestavné systémy se také často skládají z určitého počtu spolupracujících jednotek nebo procesů. A proto, je-li systém navržen tímto způsobem, je nutno jej implementovat v jazyce, který model souběžného zpracování podporuje. Právě souběžnost je klíčovou vlastností jazyka Erlang (viz kapitola 4). Procesy v něm mezi sebou komunikují zasíláním zpráv. Důležitou vlastností je i bezpečnost, ukončí-li se z nějakého důvodu jeden z běžících procesů, nijak to neovlivní chod zbývající procesů. K bezpečnosti přispívá i fakt, že čistě funkcionální konstrukce jsou bez vedlejších efektů, a tak bude jejich použitím program složen z na sobě nezávislých částí (které se navíc dají snadno modifikovat bez zásahu do částí ostatních).
Technologií, používaných pro modelaci vestavných systémů, je celá řada, například to jsou: *Concurrent Haskell* [11], *Eden* [12], *Embedded Gofer* [13], ale také jazyk *Hume* a platforma .NET Micro (jazyk C#). S posledními dvěma jsem se setkal při implementaci rozšíření kompilátoru, protože jsou jeho součástí.
Hume [9] (*Higher-order Unified Meta-Environment*) je programovací jazyk postavený na základech funkcionálního programování a byl vytvořen právě pro nasazení do vestavných systémů. Návrh softwaru v něm má být na jednu stranu vysoce abstraktní, na druhou stranu dovoluje přesný odhad časové složitosti a nároků na výpočetní výkon. Hume kombinuje funkcionální paradigma s nápady z oblasti konečných automatů. Těch je využito ke komunikaci celků v Hume, každý takový celek je pojmenován jako *box*. Vstupy a výstupy každé jednotky box jsou řešeny čistě funkcionální způsobem pomocí vysokoúrovňových struktur pattern matching. Celý jazyk Hume je složen ze tří vrstev. První je vnější (statická) deklarační vrstva, uprostřed je koordinační vrstva popisující statické rozvržení procesů a jejich zařízení, a poslední je vnitřní vrstva, která je čistě funkcionální, popisující každý proces. Hume je určen pro širokou škálu vestavných systémů, od nejjednodušších mikro kontrolérů až po mobilní telefony.
Microsoft také nabízí řešení pro vestavné systémy, a to v podobě platformy .NET Micro Framework [10]. Platforma slouží k návrhu vestavných systémů objektově orientovaným přistupem, protože v současné době je podporován pouze jazyk C#. Pro vývojáře na této platformě je jistě výhodné, že jsou pro ni nabízeny podobné nástroje jako pro vývoj běžných počítačových aplikací, například včetně vývojového prostředí Visual Studio, a jsou poskytovány zdarma. Aplikace na této platformě nepotřebuje pro svůj běh více než 300kB paměti, ani žádný operační systém. Navíc se spekuluje, že by zdrojový kód této platformy mohl být v budoucnu volně přístupný, což by mohlo vést k jejímu většímu rozšíření.

4 Erlang

První verze programovacího jazyka Erlang (podrobnější informace v [2, 3, 4, 6, 7]) vznikla v roce 1986 (byla napsána v jazyce Prolog, nynější je v jazyce C). Důvodem jeho vzniku byla potřeba společnosti Ericsson mít programovací jazyk vhodný pro použití v telekomunikacích.

4.1 Požadavky kladené na telekomunikační systémy

- Práce s velkým počtem souběžných (v jednu chvíli běžících) aktivit.
- Jednotlivé aktivity/procesy musí být vykonány v přesně určenou dobu, nebo do přesně určené doby.
- Systém je distribuován na více počítačů.
- Interakce s hardwarem.
- Velmi rozsáhlé systémy.
- Komplexní funkcionalita.
- Schopnost provozu bez přerušení po dlouhou dobu.
- Možnost údržby (upgrade, rekonfigurace, atp.) bez přerušení běhu systému.
- Zvláštní důraz na kvalitu a spolehlivost, odolnost vůči softwarovým chybám a hardwarovým selháním.

Při vývoji byly vyzkoušeny různé druhy paradigmatů: *imperative*, *declarative*, *rule based* a *object oriented*. Tým vývojářů však došel k závěru, že telekomunikační systémy nemohou být naprogramovány pouze v jednom programovacím jazyce používajícím jednu metodologii/paradigma. Erlang tak vznikl kombinací *souběžného/concurrent* (jazyky jako Ada nebo Modula) a *funkcionálního* přístupu (ML, Haskell).

4.2 Hlavní cíle při vývoji programovacího jazyka Erlang

Při vývoji jazyka byly požadovány následující vlastnosti:

- Bude postaven na virtuálním stroji, který obstarává souběžnost a správu paměti (*memory management*), což také zajišťuje nezávislost na operačním systému, a tak i přenositelnost.
- Měl by to být symbolický jazyk s garbage collectorem, dynamickým typováním.
- Měl by obsahovat datové typy jako atomy, seznamy (*lists*) a páry (*tuples*).
- Měl by obsahovat koncovou rekurzi (*tail recursion*), aby mohly být i nekonečné smyčky řešeny rekurzí.

- Měl by podporovat asynchronní přenos zpráv mezi procesy a rovněž příslušně reagovat na zprávy podle jejich druhu (pomocí *pattern matching*).
- Měl by být robustní, odolávat SW i HW chybám.

4.3 Klíčové vlastnosti jazyka Erlang

Erlang se vyznačuje následujícími vlastnostmi, které jsou pro jeho praktické využití klíčové:

4.3.1 Sekvenční Erlang (Sequential Erlang)

Tak jako i jiné funkcionální jazyky umožňuje Erlang pouze jedno přiřazení hodnoty (*single assignment*). Používá však dynamického typování. Charakteristické jsou datové typy jako: atomy, čísla, seznamy (*lists*) a páry (*tuples*). Při výběru mezi těmito datovými typy používá *pattern matching* (mechanismus, který podle podmínky vyhodnotí, jaký blok kódu se provede). Důležitým prvkem hlavně objektově orientovaných jazyků jsou smyčky (*loops*), jako například *for* nebo *while*. Pro smyčky se zde využívá pouze rekurze. Doporučuje se používat koncové rekurze (*tail recursion*) oproti běžné rekurzi, protože v koncové rekurzi se v každém kroku přiřazuje do proměnné průběžný výsledek (výsledek je proměnné přiřazen pouze jednou, v dalším kroku rekurze už se nejedná o tu samou proměnnou), na rozdíl od běžné rekurze, kde je pro výsledek nutné doběhnutí rekurze. Typicky toho můžeme využít v nekonečných smyčkách. Programy v jazyce Erlang se skládají z *modulů* (ekvivalentem v objektově orientovaném programování jsou třídy). Každý modul je zpravidla v samostatném souboru a je zvlášť zkompilován a načten. Funkce obsažené v jednom modulu mohou být zavolány z druhého modulu pouze pokud je explicitně exportujeme v modulu prvním.

```
-module(stack).
-export([push/2,pop/1,top/1,isEmpty/1]).

push(Val,{}) -> {Val,{}};
push(Val,{I1,I2}) -> {I1,push(Val,I2)}.

pop({I1,{}}) -> {};
pop({I1,I2}) -> {I1,pop(I2)}.

top({}) -> "Empty Stack";
top(S) -> top(S,0).

top({I1,{}},Top) -> I1;
top({I1,I2},Top)-> top(I2,I1).

isEmpty({}) -> true;
isEmpty({I1,I2}) -> false.
```

Výpis 1: Ukázka zásobníku (*stack*)

```
-module(quicksort).
-export([quickS/1]).

quickS([H|T]) -> quicS([X || X <- T, X =< H]) ++ [H] ++ quickS([Y || Y <- T, Y > H]);
quickS([]) -> [].
```

Výpis 2: Quicksort v jazyce Erlang

4.3.2 Souběžnost (Concurrency)

Klíčová vlastnost jazyka Erlang. Umožňuje běh více na sobě nezávislých procesů zároveň (typicky třeba telefonních hovorů). Není to vlastnost operačního systému (nejsou navrženy jako procesy operačního systému ani jako vlákna operačního systému), ale samotné implementace v jazyce Erlang. Proces je tvořen běžící funkcí, kterou je třeba vytvořit (*spawn*). Procesy mezi sebou nesdílí žádnou paměť (jsou zcela nezávislé) a mohou mezi sebou komunikovat pomocí asynchronně zasílaných zpráv (jsou vyjádřeny jakýmkoliv datovým typem podporovaným Erlangem). Zpráva je po přijetí zpracována přes *pattern matching* a proces pokračuje v běhu. Procesy jsou extrémně nenáročné a implementace jazyka Erlang podporuje systémy s velmi vysokým počtem souběžných procesů, běžně kolem 20 000 až 30 000 procesů (úspěšně byl proveden test, kdy běželo 20 miliónů procesů). Vzhledem ke skutečnosti, že systém napsaný v programovacím jazyce Erlang musí reagovat v milisekundách, nejsou možné dlouhé prodlevy *garbage collection*.

```
-module(echo).
-export([echo/0, loop/0]).

loop() -> receive
      {From, Message} -> From ! {Message};
      who_are_you -> io:format("I am ~p~n", [self()]);
      stop -> true
    end.

echo() -> NewPid = spawn(echo, loop, []),
    NewPid ! {self(), "Echo message..."},
    receive
      {Message} -> io:format("Message from echo: ~p~n", [Message])
    end,
    NewPid ! stop.
```

Výpis 3: Ukázka procesu v jazyce Erlang

4.3.3 Distribuovaný Erlang (Distributed Erlang)

Erlang obsahuje mechanismus jak pro komunikaci Erlang - Erlang, tak pro komunikaci s ostatními částmi počítače. Komunikace probíhá prostřednictvím TCP/IP socketů. Pro první případ se používá označení 'distributed Erlang system' a skládá se z více Erlang runtime systémů, jež mezi sebou komunikují. Každý takový runtime systém (Erlang

běžící na jednom počítači) se nazývá *node* a je pojmenován. Rovněž je možné se připojit k nodu, který je napsán v jazyce C (*C node* nebo *hidden node*). Ve druhém případě se jedná o komunikaci přímo jazyka Erlang s operačním systémem nebo s programy napsanými v jiných programovacích jazycích. Komunikace probíhá na bytové úrovni (příjem/odeslání byte listů) a vzhledem k tomu, že byte listy nejsou Erlangem podporovány, je potřeba naimplementovat kódovací/dekódovací mechanismus.

4.3.4 Robustnost (Robustness)

V typických aplikacích využívajících jazyk Erlang je robustnost (odolnost vůči chybám) velmi důležitou vlastností. Jestliže se proces zhroutí, nijak to nenaruší běh *nodu*, na kterém proces běžel, ani celého systému. Procesy se však mohou monitorovat navzájem, a tak se můžeme dozvědět o přerušení procesu chybovou zprávou (*error message*). Nadřazené procesy tak mohou v případě selhání procesu převzít kontrolu, zjistit příčinu, znovu nastartovat proces, atp. Tato vlastnost nám umožňuje navrhovat tzv. *soft-fail* systémy v telekomunikacích, kdy chyba v jednom telefonním hovoru může maximální přerušit tento hovor, avšak nedotkne se zbytku systému.

4.3.5 Upgrade systému bez přerušení běhu

Erlang umožňuje ugrade nebo opravu systému za chodu, aniž by musel být systém pozastaven. Jde o tzv. *hot code loading*. Nově vytvořené procesy pak již používají nový vyměněný kód, přičemž staré procesy pokračují a skončí nezávisle na novém kódu. V případě nekorektní činnosti systému stačí nahradit chybový kód za nový. Můžeme tak udržovat systém stále aktální a to bez chyb, aniž bychom ho museli pozastovavat.

```
−module(increment).
−export([newloop/0, go/0]).

go() −> loop(0).

loop(Num) −> receive
      code_switch −> io:format("Code␣replacing...~n"),
                     increment:newloop();
      {Message, Add} −> io:format("Current:␣~p~n", [Num+Add]),
                        loop(Num+Add)
        end.

newloop() −> loop(0).
```

Výpis 4: Ukázka *hot code loading*

4.3.6 Přenositelnost (Portability)

Vzhledem ke skutečnosti, že Erlang je implementován v jazyce C, je možné ho spustit všude tam, kde je tento jazyk podporován, a to jsou následující operační systémy: *Windows*, *Solaris*, *Linux* a *VxWorks*.

4.4 OTP (Open Telecom Platform)

Open Telecom Platform je vývojový systém určený pro návrh a provoz telekomunikačních systémů. OTP systém byl vyvinut společností Ericsson a v současné době je většina softwaru volně dostupná v rámci distribucí programovacího jazyka Erlang. OTP vrstva leží přímo na operačním systému a teprve na ní běží Erlang, popřípadě program v C. Součástí OTP jsou kompilátory a vývojové nástroje pro Erlang, knihovny, runtime systémy pro různá cílová prostředí (OS), vzdělávací materiály a rozsáhlá dokumentace.

4.5 Erlang v reálných aplikacích

Erlang splňuje všechny všechny výše uvedené požadavky kladené na telekomunikační systémy.

- Práce s velkým počtem souběžných (v jednu chvíli běžících) aktivit.
  → Asi 200 až 4000 zároveň aktivních procesů na přepínači *AXD 301*.
  → *Mobility Server* má kolem 200 statických procesů a pro každé volání generuje 6 dynamických procesů.

- Velmi rozsáhlé systémy.
  → *AXD 301* verze 3 obsahuje *525 KLOC Erlangu*, *608 KLOC C* a *8 KLOC Javy*. (1 KLOC je 1000 řádků kódu bez komentářů).

- Schopnost provozu bez přerušení po dlouhou dobu.
  → *Mobility Server* byl uveden na trh v roce 1994 a dnes je ve světě víc než 400 jednotek. Za tu dobu bylo hlášeno pouze několik chyb, z nichž většina byla hardwarového rázu.

- Možnost údržby (upgrade, rekonfigurace, atp.) bez přerušení běhu systému.
  → Systém v *AXD 301* sám vyřeší, jak by měl být upgradován, jak by měly být pozastaveny určité procesy, atp.

- Zvláštní důraz na kvalitu a spolehlivost.
  → *AXD 301* nepotřebuje ročně více než 6 minut zastavení chodu ze servisních důvodů.
  → GPRS má očekávanou dostupnost 99,995%, čemuž odpovídá přerušení vysílání na 26 minut za rok.

5 Programovací jazyk e-PFL

V této kapitole se budu věnovat jazyku e-PFL [1], s nímž (respektive s jeho překladačem) téma této práce úzce souvisí. Jazyk e-PFL (*Embedded Process Functional Language* nebo *Vestavný procesně funcionální jazyk*) je výsledkem snahy o vytvoření nástroje, který by umožnil modelovat vestavné systémy v prvních fázích vývoje. Je to doménově orientovaný jazyk a je určen k tvorbě vestavných systémů na vysoké úrovni abstrakce.
Funcionálních programovacích jazyků, kterých se dá využít při tvorbě nějakého systému, je poměrně dost. Vestavný procesně funcionální jazyk byl však vyvíjen s přihlédnutím k *agilním metodologiím*. Cílem agilních metodologií je snížit rizika vznikající při vývoji softwaru. Zejména jsou to *byznys rizika* (nepochopení požadavků zákazníka, a tím pádem praktická nepoužitelnost finálního softwaru) a *technická rizika* (nasazení námi zvoleného technického řešení do praxe brání nějaké vážné důvody). Agilní metodologie se těmto rizikům snaží zabránit již v prvních fázích vývoje. Z pohledu návrhu budoucího (vestavného) systému tak jde o to, co nejdříve vytvořit funkční model tohoto systému, aby mohl být předveden zákazníkovi a posouzen také z technického hlediska.
Vestavný systémy se, mimo jiné, odlišují od běžných aplikací tím, že bývají vyvíjeny na jiné platformě, než na které budou v praxi nasazeny. A tak chceme-li opravdu eliminovat technická rizika vyvíjeného systému, je potřeba abychom ho (a nebo alespoň jeho nejdůležitější části) otestovali na reálném zařízení, pro něž je navrhován. Proto i nástroj, používaný k návrhu softwaru vestavného systému, musí věrně demonstrovat běh aplikace a musí být (alespoň z části) schopen nasazení na cílové zařízení. Další věcí, která hrála roli při vývoji jazyka e-PFL, byla představa návrhu vestavného systému na takové úrovni, aby byl vývojář odproštěn od konkrétní architektury, pro níž systém vyvíjí, a nemusel řešit s ní spjaté technické detaily (obecně funcionální jazyky jsou pro vývoj vestavných systémů vhodné, protože poskytují docela vysokou míru abstrakce při psaní kódu). A tak je výsledný jazyk jakýmsi modelovacím nástrojem (čemuž přispívá také jeho syntaxe, proto je zdrojový kód snadno čitelný a pochopitelný). Kromě toho se pořád samozřejmě jedná o plnohotnotný programovací jazyk, který je schopen vyprodukovat spustitelný kód použitelný na konrétním zařízení.
Programovací jazyk e-PFL vychází z jazyka PFL. Tento jazyk stručně popíši v následující podkapitole.

5.1 PFL

PFL (*Procesně funkcionální jazyk*) je jazykem, jak už název napovídá, využívajícím procesně funcionální programovací paradigma. Je to jazyk hybridní, stojící na průniku funkcionálního a imperativního programování. Vychází z čistě funkcionální odnože Haskellu, přičemž je obohacen o možnost používat proměnné. Stav systému je tak možno popsat jednodušeji, než jak by tomu bylo při použití pouze funkcionálního jazyka (hlavně cykly a vstupně výstupní operace). Použití proměnných je řízeno kompilátorem a mohou jich využívat pouze procesy.
Proměnná v jazyce PFL není klasickou proměnnou jako v imperativních jazycích, liší se ve způsobu uložení hodnoty a přístupu k ní. Je to něco jako paměťová buňka, k níž

se přistupuje buď staticky (hodnota) nebo dynamicky (výběr a modifikace hodnoty). Programátor ale k buňce nemůže přistupovat přímo (přiřazení proměnná - hodnota), nýbrž prostřednictvím procesů. Způsoby, jakými může proces zacházet s proměnnými, jsou dva. Prvním je aplikace procesu na nějakou hodnotu, přičemž výsledek procesu je přiřazen (uložen) do proměnného prostředí, nad kterým proces pracuje a který má proces jako argument (ve tvaru *promenne_prostredi datovy_typ*, například *x Integer*). V druhým případě je proces aplikován na *řídící hodnotu* (označována jako *()*) a hodnotu z proměnného prostředí použije pro další výpočty, ale nemodifikuje ji. Proměnné, platné v aktuálním jmenném prostoru, mohou být sdíleny více procesy. Proces v PFL se od funkce odlišuje (kromě typové definice) v tom, že nemůže být předán jako argument jiné funkci, stejně tak jako nemůže být vracen jako výsledek nějaké funkce.
V jazyce PFL je možno využít jak vyhodnocování na základě hodnoty, tak vyhodnocování na základě požadavku. K dispozici je také překladač, který z jazyka PFL generuje cílový kód v jazycích Java a Haskell.

5.2 Koordinační vrstva jazyka e-PFL

E-PFL, jako i jiné funcionální programovací jazyky, má svou čistě funcionální vrstvu jazyka (jazyková úroveň) obalenou do *koordinační vrstvy*, která řeší komunikaci. Jazykové konstrukce z koordinační vrstvy se starají o vytváření procesů a komunikaci mezi nimi. Jazyk e-PFL umožňuje návrh vestavného systému, který se skládá z nějakého počtu funkčních jednotek - každá tato jednotka je reprezentována jednou proměnnou datového typu Device. Tato jazyková konstrukce je právě z koordinační vrstvy. Je to dynamická konstrukce ve smyslu jejich propojení pomocí komunikačních kanálů, které je možné v průběhu vykonávání programu měnit. Z různých důvodů však může být potřeba tyto konstrukce z koordinační vrsty nějak převést a mít k dispozici návrh celého systému na statické úrovni. Jazyk e-PFL umožňuje jakýsi kompromis, model systému na koordinační úrovni (model obsahující jazykové konstrukce z koordinační vrstvy) vzniká jako produkt vykonávání programu, je-li však nějaká funkční jednotka spuštěna, nelze její činnost již měnit a model se tak stane statickým. Tento převod z koordinační na statickou úroveň je u jazyka e-PFL v praxi řešen tak, že nejdříve je proveden *konfigurační běh*, což je částečné vyhodnocení programu v e-PFL, jehož výsledkem je znalost struktury aplikace na koordinační úrovni (konkrétní architektura vyvíjeného systému). Na základě toho jsme potom schopni generovat cílový kód (v mém případě v jazyce Erlang), který je možno nasadit na konkrétní hardware. Přičemž existuje možnost generovat cílový kód pro jeden vestavný systém ve více programovacích jazycích - jedna část systému tak může být například v jazyce Hume a druhá v jazyce Erlang.

5.3 Komunikace funkčních jednotek v e-PFL

Komunikace mezi jednotkami (Device) je v e-PFL implicitní a probíhá přes komunikační kanály, které jsou definovány předem. Spojeny jsou právě dvě jednotky (nebo třeba jednotka se standardním výstupem), přičemž každý kanál má pouze jeden vstup a pouze jeden výstup (jeden Device jím hodnotu posílá a druhý Device ji přijímá). Kanálem se

přenáší předem určený typ dat, a proto datový typ na jeho vstupu je stejný jako datový typ na jeho výstupu. V kanálu může být v jednu chvíli pouze jedna hodnota. To znamená, že poslal-li do něj vysílající Device nějakou hodnotu, nemůže do něho poslat další, dokud nebude tato hodnota zkonzumována přijímacím Device. Je také logické, že z kanálu, jenž zrovna neobsahuje žádnou hodnotu, nemůže být žádným způsobem nějaká hodnota vybrána.

5.4 Konfigurace systému v e-PFL

Jak jsem již zmínil dříve v této kapitole, program napsaný v e-PFL prochází konfiguračním během a během něho je vygenerován soubor v jazyce XML, který popisuje strukturu systému podobným způsobem, jako je popsána v e-PFL. Ve stromové struktuře konfiguračního souboru jsou jednotlivá zařízení (Device), jejich procesy a komunikační kanály. Konfigurace je klíčová pro generaci cílového kódu. Bez znalosti konfigurace daného systému bych nemohl rozšířit kompilátor, aby mohl generovat cílový kód v jazyce Erlang. V jazyce Erlang jsem sice implementoval datový typ Device, nicméně tento návrh je víceméně abstraktní a je třeba ho pro každý systém zkonkretizovat (počty zařízení, jejich propojení, atd.) tak, aby byl funkčně ekvivalentní návrhu v jazyce e-PFL. Tuto konkretizaci mi umožňuje právě konfigurační soubor, ve kterém jsou všechny potřebné informace o daném systému uloženy. Praktické realizaci rozšíření kompilátoru se věnuji v kapitole 6.
Nyní bych ukázal příklad zdrojového kódu v jazyce e-PFL a k němu odpovídající konfiguraci. Tato část kódu v e-PFL obsahuje definici jedné proměnné datového typu Device - proměnná s názvem *work* (předposlední řádek ve výpise). Toto zařízení v sobě obsahuje dva procesy - *multiply* a *suma*, které jsou, jako u každého zařízení, umístěny ve stromové struktuře (poslední řádek). Proces multiply, stejně jako proces suma, bere jeden argument a jako výsledek vrací trojici čísel (řádky 1 a 2, respektive 3 a 4).

```
multiply :: a Integer -> (a Integer, b Integer, c Integer)
multiply x = (x+1, y, y) where {y = x * x;}
suma :: a Integer -> (a Integer, b Integer, c Integer)
suma x = (x+1, y x, y x) where {y a = compute a (a+x);}
work :: Device
work = Fair[Process multiply,Process suma]
```

Výpis 5: Ukázka zdrojového kódu v jazyce e-PFL

Následující výpis kódu v jazyce XML je tedy konfigurace výše popsaného zařízení. Jelikož je celá konfigurace poměrně rozsáhlá, proto zde uvádím pouze její část, dobře vyjadřující princip konfigurace. Hned první tag ve výpise se jmenuje Fair a určuje kořen stromové struktury. Zařízení z příkladu obsahuje dva vestavné procesy - v konfiguraci jsou vymezeny tagy Process. Dále jsou ve struktuře vidět jména procesů (tag EmbeddedProcess a jeho atribut name) a také názvy vstupů a výstupů každého procesu (ohraničeny tagy Input respektive Output).

```
<Fair>
  <Process>
  <EmbeddedProcess name="multiply">
    <Input>
      <EmbeddedVariable ref="a" />
    </Input>
    <Output>
      <EmbeddedVariable ref="a" />
      <EmbeddedVariable ref="b" />
      <EmbeddedVariable ref="c" />
    </Output>
  </EmbeddedProcess>
  </Process>
  <Process>
  <EmbeddedProcess name="suma">
    <Input>
      <EmbeddedVariable ref="a" />
    </Input>
    <Output>
      <EmbeddedVariable ref="a" />
      <EmbeddedVariable ref="b" />
      <EmbeddedVariable ref="c" />
    </Output>
  </EmbeddedProcess>
  </Process>
</Fair>
```

Výpis 6: Ukázka konfigurace systému

Konfigurace každého zařízení dále obsahuje jeho jednoznačné pojmenování v rámci celého systému a popis vstupů a výstupů pro celé zařízení (nemusí korespondovat se vstupy a výstupy jednotlivých procesů). Právě tyto vstupy a výstupy jsou viditelné v celém systému a přes ně mohou být zařízení propojena. V konfiguračním souboru jsou udána jejich jména a datové typy (jiné datové typy pak kanálem neprojdou). Komunikační kanály je také možno přejmenovat (tag Annotation) nebo jim nastavit výchozí hodnotu (tag InitialValue).

6 Praktická realizace rozšíření kompilátoru

V této části se zaměřím na popis praktické části této bakalářské práce, jejímž cílem bylo rozšíření již existujícího kompilátoru, který je jedním z nástrojů vyvinutých pro práci s jazykem e-PFL. Jak již jsem dříve uvedl, tento kompilátor byl doposud schopen překladu e-PFL do dvou cílových kódů - funkcionálního jazyka Hume a do jazyka C# na platformě .NET Micro Framework.
V první fázi bylo nutné se seznámit se syntaxí jazyka e-PFL, ale hlavně s konstrukcemi pro tento jazyk specifickými. Jeho syntaxe se odlišuje více od syntaxe jazyka Erlang, než od jiných funkcionálnách jazyků (jako Haskell). Hlavní rozdíly, co se syntaxe týče, jsou v tom, že e-PFL používá statické typování (programátor určuje datový typ proměnných), naproti tomu Erlang používá dynamické typování (programátor neurčuje datové typy), takže kód napsaný v jazyce Erlang je na pohled ještě stručnější. Navíc Erlang nepoužívá uživatelsky definované datové typy, což souvisí s tím, že je to dynamicky typovaný jazyk, v čemž se opět liší od e-PFL (nicméně od verze R14B bude Erlang oficiálně obsahovat rozšíření, které bude uživatelské datové typy podporovat).
Výše uvedené rozdíly, jakož i běžné rozdíly v syntaxi, však nebyly hlavním problémem při překladu. Tím byl datový typ *Device*, který je pro jazyk e-PFL, potažmo pro návrh vestavných systémů, podstatný, a který v programovacím jazyce Erlang nemá ekvivalent. Proto v následující podkapitole popíši, jakým způsobem jsem tento datový typ implementoval do jazyka Erlang (jako dokumentaci jsem využíval [2, 6]). Všechny zdrojové kódy v jazyce Erlang jsem psal v programu *Poznámkový blok*, přičemž k jejich kompilaci a spouštění jsem používal *Erlang emulator* ve verzi 5.7.4.

6.1 Datový typ Device

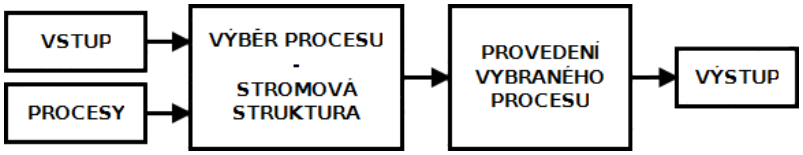
Datový typ Device v jazyce e-PFL je reakcí na skutečnost, že vestavné systémy (ať už s jedním nebo více procesory) jsou často modelovány jako soubor spolupracujících jednotek. Právě datový typ Device simuluje jednu takovou jednotku. Každá taková jednotka (dále jen Device nebo zařízení) obsahuje nějaký počet *procesů*, to znamená, že nemusí nutně vykonávat pouze jednu činnost. Proces (takto se tato část zařízení označuje pouze v terminologii jazyka e-PFL, nejedná se o proces jak jej známe např. z jazyka Erlang) už je v podstatě běžná funkce. Každý Device má nějaký počet vstupů, které jsou následně dány ke zpracování některému z procesů. Počet vstupů u každého zařízení je stejný, jako největší arita ze všech jeho procesů. V danou chvíli nemusí být k dispozici všechny vstupy. Je již na programátorovi, jaké procesy (s jakou aritou) zařízení obsahuje, a tím pádem, které vstupy jsou zkonzumovány a které (případně) nikoliv. Pokud má Device v danou chvíli k dispozici takový počet vstupů, že jim nevyhovuje žádný proces, automaticky čeká na další vstupy (kanály ve kterých už vstupy jsou se zablokují), aby mohl být nějaký proces proveden.
Device má všechny svoje procesy uspořádané ve stromové struktuře, ve které, po přijetí nějakého vstupu, hledá vyhovující proces (porovnává arity jednotlivých procesů s počtem vstupů, jež přijal), který bude vykonán (a proto to, jestli bude vykonán, nezávisí pouze na jeho aritě). Strom se skládá z uzlů, přičemž každý má jednu *fairness* hodnotu,

a to buď *fair* nebo *unfair*, a z listů, což jsou jednotlivé procesy. Na každém uzlu může být libovolný počet listů nebo jiných uzlů, navíc listy nemusí být na nejspodnější vrstvě stromu. Jediným omezením je, že list nemůže být spojený s dalším listem.
Po nalezení procesu se na strom uplatňuje algoritmus, který slouží ke změně jeho struktury. Jak se struktura změní (a jestli vůbec) záleží na tom, jakými uzly vyhledávací algoritmus prošel (vždy projde alespoň jedním uzlem). Platí-li, že uzel, ležící na cestě k nalezenému procesu, má hodnotu unfair, pořadí podstromů [1] z něch vycházejících se nemění. Má-li však uzel hodnotu fair, potom se celý podstrom, který proces obsahuje, přesune na konec té vrstvy stromu, kde tento fair uzel je. Tj. doprava, pokud by byl strom znázorněn s kořenem nahoře a větve by šly dolů, nebo doleva, v opačném případě.
Tento mechanismus řeší problém při vybírání ze dvou (a více) procesů se stejnou aritou v rámci jednoho zařízení. Pokud tedy jsou procesy se stejnými počty argumentů připojeny na uzel s hodnotou fair, ve vykonávání se tyto procesy pravidelně střídají (vykonaný proces je vždy posunut nakonec). Struktura stromu záleží na programátorovi, který daný vestavný systém navrhuje, a typicky nebude moc složitá. Při chybném návrhu stromu se může stát, že některý proces nebude nikdy vybrán (pokud by to byl úmysl, pak je zbytečné, aby zařízení tento proces vůbec obsahovalo).
Pro potřebu vzájemné komunikace je možné jednotlivá zařízení spojit jednosměrným komunikačním kanálem, přičemž je také možné spojit zařízení samo se sebou. V praxi toto spojení funguje tak, že je-li vykonán proces uvnitř jednoho zařízení, je následně jeho výsledek poslán tímto komunikačním kanálem k připojenému zařízení. Tato zpráva mu přijde na vstup a výše popsaný proces fungování zařízení se opakuje.

6.2 Implementace datového typu Device v programovacím jazyce Erlang

V následující podkapitole popíši, jakým způsobem je implementován datový typ Device v jazyku Erlang. Na začátku bylo důležité navrhnout strukturu tohoto datového typu podle jeho funkčnosti v e-PFL. Nezáleží na tom, jestli se dvě praktické implementace jedné a též věci liší, důležitá je výsledná ekvivalentní funkčnost těchto implementací. Device slouží k popisu jedné funkční jednotky vestavného systému a má určitou funkčnost, kterou je samozřejmě nutné dodržet, proto aby návrh vestavného systému, typicky složený z více takových funkčních jednotek, napsaný v jazyce e-PFL měl stejnou funkčnost jako vygenerovaný cílový kód, který bude nasazen na konkrétní hardware.
Jelikož Device musí dodržovat určité vlastnosti a mít určitou funkčnost, bylo v jazyce Erlang potřeba celou jeho strukturu rozdělit na několik menších, spolupracujících částí - funkcí. Chtěl jsem, aby byl výsledný kód co nejkratší a nejsrozumitelnější a aby byl jakýmsi logickým přepisem nějaké teoretické samostatné funkční jednotky (ne nutně té z e-PFL), jednotky, která má nějakou vnitřní funkčnost a dokáže komunikovat s okolím. Datový typ Device jsem si proto představil jako kontejner, obsahující daný počet vestavných procesů (respektive funkcí). Jak jsem již uvedl výše, vestavné procesy jsou uspořádány a vybírány ze stromové konstrukce, proto i tento kontejner obsahuje mechanismus, v

[1] Podstromem je zde myšlena stromová struktura, uzel bez dalších vazeb nebo list (a to i ten, reprezentující vybraný proces)



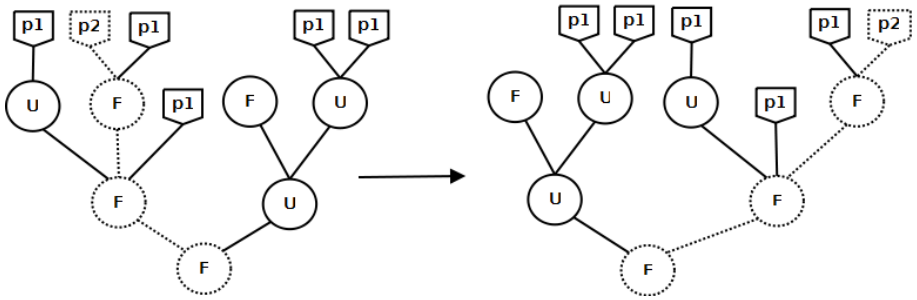
Obrázek 1: Schéma zařízení

němž je strom držen, jakož i algoritmus potřebný pro změnu jeho struktury po vybrání jednoho z procesů, který je v něm umístěn. Komunikaci mezi zařízeními (jak jejich vstup tak i výstup) jsem řešil tak, jak komunikace standardně probíhá v programovacím jazyce Erlang - pomocí zpráv zasílaných mezi procesy. Každá výše uvedená funkční část jednoho zařízení je v mé implementaci reprezentována jako samostatný proces jazyka Erlang (k čemuž se podrobněji dostanu dále v této kapitole). Zařízení ilustruje schéma na obrázku 1.

Typ Device je v jazyce Erlang implementován jako soubor několika funkcí, respektive procesů. V následujících podkapitolách bude každá funkce, nezbytná pro funkčnost, popsána.

6.2.1 Reprezentace stromu s procesy

Již dříve jsem poukazoval na důležitost a smysl, proč jsou procesy v rámci jednoho zařízení umístěny ve stromové struktuře. Při implementaci jsem proto musel určit, jakým konkrétním způsobem má být tato struktura reprezentována. Ve funkcionálních jazycích je v důsledku strom běžně vyjádřen většinou jako seznam seznamů. Tato struktura se mi, pro dané použití, jevila jako vyhovující, nicméně bylo ji třeba obohatit tak, aby později mohla obsahovat všechny potřebné informace.
Strom s procesy typicky nebývá nějak složitě rozvětvený a ani nebývá moc vysoký, nicméně jeho struktura může být velice nepravidelná, nejedná se o binární ani o jiný druh stromu s určenou strukturou. U první struktury, kterou jsem navrhl, jsem vycházel ze skutečnosti, že ve stromě se vyhledávají procesy - listy, a že uzly stromu v tuto chvíli nejsou důležité. Navíc, v rozumně navrženém stromě, jsou listy s procesy ve stromě nejníž. Strom byl proto vyjádřen jako seznam seznamů, přičemž každý seznam obsahoval prvky (listy i uzly) z jedné vrstvy stromu. Nejníž položená vrstva stromu byla v seznamu jako první, ta nejvýš položená byla poslední. U každého prvku také bylo uvedeno číslo určující pořadí prvku z vyšší vrstvy, ke kterému je prvek připojen. Vyhledávací algoritmus byl tak poměrně jednoduchý a pro většinu stromů také efektivnější než algoritmus, který by stromem procházel shora. Problém ovšem nastal po vyhledání, kdy bylo potřeba změnit strukturu stromu. Řešení by bylo zbytečně obtížné a neefektivní, proto jsem vymyslel jinou reprezentaci stromu, ve které se poměrně jednoduše dá vyhledávat, stejně jako měnit její strukturu (příklad takové operace je na obrázku 2).
Nynější reprezentace vyjadřuje strom shora dolů a každý seznam je podstromem. Na rozdíl od přechozí implementace se tak jedná o rekurzivní datovou strukturu.



Obrázek 2: Změna struktury stromu po provedení procesu *p2*

```
T1 = [ fair , [ unfair , unfair , {p2, 2}, {p1, 1}], [ fair , {p1, 1}, {p2, 2}]].
T2 = [ fair , [ unfair , unfair , {p2, [a, b]}, {p1, [c ]}], [ fair , {p1, [c]}, {p2, [a, b ]}]].
```

Výpis 7: Implementace stromu

Vždy prvním prvkem v seznamu je uzel, jeho datový typ je atom a může nabývat hodnot fair a unfair. Je to zároveň i rodič pro všechny prvky následující za ním. List s procesem je ve stromě jako pár (*tuple*, ve složených závorkách), kde jeho první hodnota je název vestavného procesu a druhá hodnota je jeho arita. Strom T2 je stejný až na to, že místo arity procesu obsahuje seznam s názvy jeho vstupů.

6.2.2 Pomocné funkce

Na každou proměnnou datového typu Device v původním návrhu systému v jazyce e-PFL připadají ve vygenerovaném cílovém kódu v jazyku Erlang čtyři funkce emulující její funkčnost. A tak, překládá-li se návrh systému, obsahující například 5 proměnných typu Device, v cílovém kódu v jazyce Erlang už je 20 funkcí - pět krát čtyři principiálně stejné funkce (samozřejmě s rozdílnými názvy, korespondujícími s názvy zařízení, jejichž částmi jsou). Tyto funkce potřebují pro svou činnost pomocné funkce. Ty jsou v cílovém kódu pouze jednou, nemají vazbu na konkrétní zařízení a slouží pouze k návratu hodnot, potřebných pro správné fungování jednotlivých zařízení.
První z těchto funkcí je funkce search (zdrojový kód je uveden v příloze), která vyhledává proces ve stromě. Je napsána tak, že je možné proces vyhledávat podle jeho jména (toho jsem využíval při testování), podle arity (první verze implementace datového typu Device) nebo podle seznamu se jmény vstupů (nynější implementace). Funkce vrací jméno procesu, který bude zavolán. Jako argumenty má funkce strom samotný (dvakrát, jedním prochází a druhý potom předává funkci, která změní jeho strukturu), hledanou hodnotu, název procesu, držícího strom pro celé zařízení (proces funkce tree_holder), název procesu, jež je zodpovědný za volání procesů uvnitř zařízení a jemuž proto bude zaslán název vybraného procesu (proces funkce process_caller) a seznam, pojmenovaný jako Position_list , do něhož funkce ukládá cestu, kterou postupně prochází, než najde hledaný proces. Tento seznam se také předává funkci, která mění strukturu stromu. Algoritmus funguje tak, že postupně prochází seznam (nějakou část stromu) a hledá pár vyhovující

hledanému procesu. Každý seznam obsahuje alespoň jeden atom a nějaký počet dalších seznamů a párů. Po každém přesunutí na další prvek je inkrementována aktuální pozice v seznamu Position_list . Jestliže není list s procesem nalezen v aktuální seznamu, funkce rekurzivně vstoupí do prvního neprozkoumaného seznamu (podstromu) a zároveň je do seznamu Position_list přidána další položka značící další vrstvu stromu. Nelze-li už jít ve stromě níž, funkce vyskočí o úroveň výš a vstoupí do dalšího podstromu v pořadí. V tomto ohledu se algoritmus chová jako většina jiných rekurzivních implementací pro průchod stromem. Algoritmus prochází stromem zleva doprava (při znázornění stromu s kořenem nahoře), a tak, pokud by byl použit na binární vyhledávací strom, vracel by hodnoty listů od nejmenších po největší.
Tato funkce je jmenuje shuffle a mění strukturu stromu podle vzoru, který jsem popisoval dříve v této kapitole. Funkce vrací změněný strom. Jako argumenty bere stromovou strukturu, jíž má změnit a seznam, ve kterém jsou uzly, ležící na cestě k vybranému procesu, a list s vybraným procesem. Hodnoty v tomto seznamu však neznamenají pouze cestu k vybranému procesu, ale označují i prvky v jednotlivých vrstvách stromu, které mají být v rámci své vrstvy případně posunuty na konec, aby byl strom korektně přestavěn. Algoritmus prochází stromem shora a vždy nejdřív zjišťuje, jaká je první hodnota v seznamu, ve kterém se právě nachází (jinými slovy zjišťuje fairness hodnotu aktuálního uzlu), podle toho určený prvek posune/neposune a následně do stejného prvku běh funkce vstoupí (rekurzivní volání) a ze seznamu s cestou odebere první hodnotu (vrstvu stromu, ze které právě sestoupil). Algoritmus skončí ve chvíli, kdy je seznam s cestou prázdný, a tím pádem je proměna stromu dokončena.

```
shuffle(Tree, []) -> Tree;
shuffle(Tree, [{fair, Pos}|T]) ->
 lists:sublist(Tree, Pos - 1) ++ lists:nthtail(Pos, Tree) ++ [shuffle(lists:nth(Pos, Tree), T)];
shuffle(Tree, [{unfair, Pos}|T]) ->
 lists:sublist(Tree, Pos - 1) ++ [shuffle(lists:nth(Pos, Tree), T)] ++ lists:nthtail(Pos, Tree).
```

Výpis 8: Funkce shuffle

Další pomocná funkce se jmenuje return_item_at a vrací vždy první neprozkoumaný podstrom (seznam) aktuální vrstvy stromu, do kterého se následně přesune běh funkce. Index, označující tento prvek bere ze seznamu Position_list . Funkce return_item_at je vždy volána až potom, co funkce search prošla celý podstrom a nenašla vyhovující list s procesem.
Funkce increase_position_counter slouží k inkrementaci hodnoty v seznamu Position_list , vždy po přeskočení na další prvek ve stromě. Jelikož Position_list není jednoduchý seznam (má tvar [{a, b}, {c, d}]), proto jsem z důvodu čitelnosti, přehlednosti, a tím pádem také menší náchylnosti k chybám, raději tuto operaci napsal do samostané metody, než abych ji vždycky prováděl přímo v poli pro argumenty funkce search.
Následující pomocnou funkcí je funkce output_creator. Ta je volána vždy až na konci jednoho cyklu celého zařízení (po provedení vybraného vestavného procesu a před odesláním této hodnoty připojenému zařízení) a slouží k tomu, že výsledek vykonaného procesu je přiřazen k názvu příslušného komunikačního kanálu tak, aby mohla být hodnota správně přijata v napojeném zařízení. Funkce najednou prochází seznamy s výsledky a s

názvy komunikačních kanálů (tento seznam je načten z konfigurace systému), prvky na stejných pozicích slučuje do páru, ten pak připojí do výsledného seznamu ([{jmeno_kanalu, hodnota}, {jmeno_kanalu, hodnota}]), který už je odeslán připojenému zařízení. Argumenty funkce jsou tedy tři seznamy - s výsledky procesu, s názvy komunikačních kanálů a výsledný seznam, jenž bude odeslán.

```
output_creator ([],   [],   Res) -> Res;
output_creator ([],   [Vh|Vt],  Res) -> Res;
output_creator ([Nh|Nt],  [],   Res) -> Res;
output_creator ([Nh|Nt],  [Vh|Vt],  Res) -> output_creator(Nt, Vt, Res ++ [{Nh, Vh}]).
```

Výpis 9: Funkce output_creator

Funkce extract slouží k vybrání požadovaných hodnot ze vstupu, který zařízení obdrží. Vstup má vždy podobu seznamu obsahujícího páry. V každém páru je název komunikačního kanálu (respektive jeho vstupu do aktuálního zařízení) a hodnota, kterou tento kanál právě obsahuje a která byla nyní doručena do zařízení. Pro další zpracování je potřeba oddělit tyto dvě složky a mít samostatně dva seznamy - jeden pouze s názvy kanálů a druhý pouze s hodnotami (samozřejmě ve stejném pořadí). Funkce tedy prochází přijatý seznam a vybírá pouze potřebné hodnoty, které ukládá do samostaného seznamu. Po průchodu celým doručeným seznamem je jako výsledek vrácen seznam nový. Funkce má tedy jako argumenty procházený seznam, kam ukládá vybírané hodnoty, a atom značící, které hodnoty má funkce vybírat (name nebo value).
Další pomocná funkce, get_device, slouží k vyhledání konkrétního zařízení v seznamu obsahujícím všechna zařízení a návracení identifikátoru jeho procesu. Tento seznam má tvar ev_name, PID[d, dev_name, PID] a je funkci předán jako argument společně se jménem zařízení, které má najít. Funkce jednoduše prochází seznam a u každého páru porovnává jeho první hodnotu s vyhledávaným jménem. Shodují-li se, funkce vrací druhou hodnotu tohoto páru - identifikátor hledaného procesu.

6.2.3 Spouštění zařízení

Jak jsem uvedl již dříve, každé zařízení je v rámci konfigurace celého systému jednoznačné pojmenováno. Právě tato jména jsou využita k tomu, aby byly jednotlivé funkce každého zařízení od sebe ve vygenerovaném kódu odlišeny. A tak všechny funkce náležící jednomu zařízení mají název ve tvaru zarizeni_funkce (například device0_input_box), přičemž funkce spouštějící zařízení se jmenuje právě jako zařízení, které spouští. Tato funkce nebere žádné argumenty (například device0()). Tato funkce v podstatě slouží pouze ke spuštění dvou procesů, které ve své podstatě tvoří hlavní funkcionalitu zařízení, a její návratovou hodnotou je identifikátor procesu (v jazyce Erlang označovaný jako *PID*), který zastupuje celé zařízení a komunikuje s ostatními zařízeními.

```
dev0() -> T_process = spawn(erlang_code, dev0_tree_holder, [[fair, {multiply ,[e]},  {suma,[e]}]]),
          I_process = spawn(erlang_code, dev0_input_box, [T_process]),
          I_process.
```

Výpis 10: Spouštění zařízení

Jak je vidět z výpisu, v těle funkce jsou inicializovány dvě hodnoty, jimiž jsou T_process a I_process. T_process označuje proces, který vznikl spuštěním funkce tree_holder, jíž byl zároveň předán strom s procesy celého zařízení jako argument. Proces I_process je spuštěn z funkce input_box s argumentem T_process - tím pádem je zajištěna interakce mezi stromovou strukturou s procesy a zbytkem zařízení. A na posledním řádku funkce je identifikátor procesu I_process vrácen jako výsledek, s nímž už je možné komunikovat.

6.2.4 Propojení zařízení

V e-PFL jsou zařízení propojena pomocí komunikačních kanálů. Každé zařízení má definovaný určitý počet vstupů (input) a výstupů (output) a každý z nich je jednoznačně pojmenován. Podle toho se proto dá určit, která dvě zařízení jsou spolu propojena. Název výstupu jednoho zařízení je stejný jako název vstupu k němu připojenému zařízení. Dá se tak realizovat i propojení více než dvou zařízení současně (kdy jedno zařízení má třeba dva výstupy a každý je připojen k jinému zařízení). V e-PFL není proto explicitně definován vztah jako *devX je připojeno k devY*. Tato varianta by samozřejmě byla jednodušší na implementaci v jazyce Erlang, jelikož v něm reprezentuji každé zařízení jako jeden proces, a tak bych komunikaci vyřešil pouze jako zasílání zpráv mezi procesy, přičemž by ani nebylo nutné provádět jakoukoliv kontrolu vstupu, respektive výstupu, protože by bylo jasné, že spolu musí souhlasit. Když jsem přemýšlel, jaký mechanismus propojení všech zařízení bude nejvhodnější, uvědomil jsem si, že celý vestavný systém má ze softwarového hlediska strukturu podobnou počítačové síti. Jelikož komunikačních kanálů může být teoreticky libovolně mnoho, bylo by nešikovné explicitně definovat každé konkrétní spojení. Stejně tak v počítačové síti není každý počítač přímo připojen se všemi dalšími počítači, ale je využíváno síťových prvků, které komunikaci řídí.
Do kódu jsem proto přidal funkce, které fungují na podobném principu jako *přepínač* (*switch*) - starají se o posílání zpráv mezi zařízeními. Proto je potřeba znát identifikátory PID všech zařízení v kontextu celého systému, ten se proto startuje funkcí run(), ve které nejdřív dojde ke spuštění všech zařízení a uložení jejich identifikátorů, společně s názvy zařízení, do seznamu. Ten je následně předán jako argument při spouštění procesu funkce sw_receiver. PID tohoto procesu, reprezentující náš pomyslný přepínač, je potom funkcí init rozeslán všem zařízením. To proto, že výstupy všech zařízení budou odesílány právě jemu, a až potom ke konkrétním zařízením.

```
run() -> Dev_list = [{dev0, spawn(erlang_code, dev0, [])},{dev1, spawn(erlang_code, dev1, [])}],
S = spawn(erlang_code, sw_receiver, [Dev_list]) ,
 init ( Dev_list , S).
```

Výpis 11: Spuštění systému

Funkce init je velmi jednoduchá. Pouze prochází seznam Dev_list a každému zařízení zprávou pošle PID procesu funkce sw_receiver.
Mechanismus přepínače se skládá ze dvou funkcí. První je již výše zmíněná sw_receiver, druhou je funkce switch. Sw_receiver slouží k přijímání zpráv ze zařízení a navíc v sobě drží seznam Dev_list. Každá zpráva (seznam) obsahuje počet trojic (shodný s počtem výstupů zařízení, které zprávu odeslalo), kde každá obsahuje název komunikačního

kanálu, jeho hodnotu a jméno zařízení, do něhož tento kanál vstupuje, například [{dev0, a, 3}, {dev0, b, 21}, {dev1, c, 1}]. Zpráva je hned, jako argument, předána funkci switch spolu se seznamem procesů zařízení a s atomem send, což znamená, že funkce switch bude jednotlivé hodnoty rozesílat příslušným zařízením. Následně se funkce sw_receiver znovu volá, z důvodu uchování seznamu Dev_list.
Funkce switch se již stará o samotné rozesílání hodnot zařízením. To znamená, že prochází přijatým seznamem, z každé trojice zjistí název zařízení, kterému se má hodnota doručit, pomocí funkce get_device (popsána výše) zjistí ze seznamu Dev_list jeho PID, na který odešle hodnotu a také název kanálu. Až jsou všechny hodnoty rozeslány (z přijaté zprávy zbyl prázdný seznam), funkce switch vstoupí do své druhé klauzule (nyní s argumentem done), která má za úkol všem zařízením poslat zprávu [done], na znamení toho, že všechny hodnoty již byly rozeslány, a zařízení tak mohou začít s jejich zpracováním - nemůže se proto stát, že zařízení začne zpracovávat vstup a najednou mu přijde ještě nějaká hodnota.

```
sw_receiver(Dev_list) -> receive
                            List -> switch(send, List, Dev_list)
                        end,
                        sw_receiver(Dev_list).
```

Výpis 12: Funkce sw_receiver

```
switch(done, []) -> done;
switch(done, [H|T]) -> element(2, H) ! [done], switch(done, T).

switch(send, [], Dev_list) -> switch(done, Dev_list);
switch(send, [{Dev_name, Name, Value}|T], Dev_list) ->
get_device(Dev_name, Dev_list) ! [input, {Name, Value}], switch(send, T, Dev_list).
```

Výpis 13: Funkce switch

6.2.5 Příjem zpráv

Funkce input_box je část zařízení, která se stará o přijetí zpráv z funkce switch. Obsahuje běžnou konstrukci pro příjem zpráv v programovacím jazyce Erlang (blok *receive - end*), která obsahuje tři šablony. První z nich, [init , Sw], je využita pouze jednou, a to hned na začátku, kdy přijme zprávu odeslanou z funkce init , díky které má od té chvíle zařízení platný identifikátor PID mechanismu přepínače (předtím má tento argument hodnotu null). Šablona [input, New_input] slouží pro příjem vstupů z přepínače. Poslední šablona ve tvaru [done] indikuje, že příjem zpráv může být pozastaven a je možné zpracovat přijaté hodnoty. Za každou klauzulí volá funkce sebe samu, samozřejmě s pozměněnými argumenty. Těmi jsou: aktuálně přijaté hodnoty, PID procesu držící stromovou strukturu zařízení a PID přepínače. V případě šablony [input, New_input] se k dosavadním přijatým vstupům (uloženým v seznamu) připojuje hodnota New_input. Po ukončení příjmu ([done]) se ještě před novým zavolání funkce sebe samé (pro kumulaci vstupů je teď logicky předán prázdný seznam) spustí proces funkce process_caller (popis dále v této kapitole), který je, spolu s dalšími hodnotami, hned poslán procesu obsahující stromovou strukturu,

ten se postará o vyhledání vestavného procesu, který bude proveden, na základě hodnot přijatých tady, ve funkci input_box.

```
input_box(Input, Tree, Switch) ->
receive
    [ init , Sw] -> input_box([], Tree, Sw);
    [done] -> C = spawn(module, process_caller, [Switch, extract(value, Input, []) ]) ,
              Tree ! [search, extract(name, Input, []) , C],
              input_box ([], Tree, Switch);
    [input, New_input] -> input_box(Input ++ [New_input], Tree, Switch)
end.
```

Výpis 14: Funkce input_box

6.2.6 Výběr vestavného procesu

Všechny operace nad stromem s vestavnými procesy, hlavně výběr procesu, jsou řízeny funkcí tree_holder, respektive jejím procesem, který je jedním ze dvou pořád běžících procesů (spolu s procesem funkce input_box - popis výše) reprezentujících jedno zařízení. Jako jediný argument této funkce je již zmíněný strom. Funkce má za úkol čekat na zprávu, indikující, jakou operaci bude potřeba provést, a to buď vyhledání stromu nebo jeho aktualizaci. Jako první se vždy provede vyhledávání - zpráva ve tvaru [search, Args, Process_caller], která je odeslána z procesu funkce input_box. Kromě atomu search a identifikátoru procesu funkce process_caller, který zodpovídá za samotné provedení vestavného procesu, zpráva obsahuje seznam s názvy kanálů (Args), ve kterých je právě přítomna hodnota. Tento seznam vznikl použitím funkce extract na seznam s přijatými vstupy. Tento seznam bude nyní ve stromě vyhledáván. A tak je zavolána funkce search a je jí předán strom v aktuální podobě, právě přijatý seznam s názvy kanálů, vlastní PID (funkce self()), to aby se později vědělo, kde má být poslán přestavěný strom, právě přijatý PID procesu process_caller, prázdný seznam pro ukládání cesty a atom new_level. Po nalezení jména procesu ve stromě je z funkce search poslána příslušná zpráva procesu process_caller s informací, který vestavný proces byl vybrán, potom je na strom volána funkce shuffle a nakonec je přestavěný strom zpátky poslán tomuto procesu tree_holder, a to jako zpráva [update, Updated_tree]. Aktualizovaný strom Updated_tree je hned předán jako argument (proces funkce volá sám sebe), což zaručuje, že proces tree_holder vždy obsahuje nejnovější strukturu stromu.

```
tree_holder(Tree) ->
receive
   [search, Args, P_caller] -> search(Tree, Tree, Args, self(), P_caller, [], new_level),
                               tree_holder(Tree);
   [update, Updated_tree] -> tree_holder(Updated_tree)
end.
```

Výpis 15: Funkce tree_holder

6.2.7 Provedení vestavného procesu

Poslední funkcí, která je součástí mé implementace datového typu Device, je funkce process_caller. Funkce slouží pro příjem zpráv odesílaných z procesu funkce tree_holder, respektive z funkce search, které říkají, jaký vestavný proces [2] má být vykonán. Funkce obsahuje tolik šablon pro příjem zpráv, kolik je v zařízení vestavných procesů, a každá šablona tak odpovídá jednomu vestavnému procesu. Zprávy i šablony jsou ve tvaru [call_process, Embedded_process_name]. Za každou šablonou už následuje samotné zavolání vestavného procesu. Případné argumenty vestavného procesu jsou funkci process_caller známy a sama je má jako argumenty, které dostala když byl spouštěn její proces ve funkci input_box, a které byly z přijatého seznamu, jež byl odeslán z přepínače, získány pomocí funkce extract, proto jsou vestavnému procesu automaticky předány (shodují se názvy jednotlivých hodnot).
Výsledek vestavného procesu je ve formě seznamu předán funkci output_creator, která k jednotlivým hodnotám přidá jména příslušných komunikačních kanálů, stejně jako názvy zařízení, do nichž tyto kanály vstupují (tyto informace jsou při generaci cílového kódu známy z konfigurace systému), čímž vznikne zpráva ve tvaru, který již dokáže zpracovat přepínač, a je mu proto poslána (PID přepínače má funkce process_caller jako svůj první argument).

```
process_caller(Sw, [E]) ->
receive
[call_process, multiply] -> Tmp = multiply(E),
                            Sw ! output_creator([{dev1, b},{dev2, d},{dev0, e}], Tmp, []);
[call_process, suma] -> Tmp = suma(E),
                            Sw ! output_creator([{dev1, b},{dev2, d},{dev0, e}], Tmp, [])
end.
```

Výpis 16: Funkce process_caller

Tímto krokem je ukončen jeden cyklus celého vestavného systému a běh programu je znovu přesunut do mechanismu přepínače.

6.3 Rozšíření kompilátoru

V této části popíši průběh rozšiřování kompilátoru, který je pravděpodobně nejdůležitějším nástrojem pro práci s jazykem e-PFL a slouží k překladu v e-PFL napsaném návrhu systému do některého z cílových jazyků. Kompilátor je implementován v jazyce C# na platformě .NET ([8]) a k programování jsem používal integrované vývojové prostředí *Microsoft Visual Studio 2008*.
Výchozí pozice, z níž má implementace začínala, byla v bodě, kdy již zdrojový kód v e-PFL prošel lexikální analýzou, parsováním (syntaktická a sémantická analýza) a byla vyprodukována konfigurace v jazyce XML. Věděl jsem tedy, jak daný vestavný systém vypadá, to znamená: kolik obsahuje zařízení, jakými komunikačními kanály jsou mezi sebou zařízení propojena a jaké vestavné procesy každé zařízení obsahuje. S použitím

[2] Vestavný proces je jak v mé implementaci, tak i původně v e-PFL vyjádřen jako běžná funkce, a s procesem v pravém slova smyslu nesouvisí

těchto dat a výše popisovaného mechanismu každého zařízení již bylo možné vytvořit zdrojový kód, popisující konkrétní zařízení, v jazyce Erlang.

6.3.1 Staticky definované funkce

Funkce, které budou vždy součástí vygenerovaného kódu a které nijak nezávisí na struktuře vestavného systému, jsou uloženy v souboru *ErlangIncludes.ctp*. Jedná se o pomocné funkce popisované v předchozí podkapitole, jsou využívány všemi zařízeními pro různé menší operace. Celý obsah tohoto souboru je při generaci vložen do cílového zdrojového kódu. V souboru *ErlangPrimitives.ctp* jsou navíc, pro případ potřeby (běžně se do cílového kódu nevkládají), definované primitivní funkce pro operace jako sčítání, odčítání, atp.

6.3.2 Erlang Generator

Celá logika, starající se o generaci cílového kódu v programovacím jazyce Erlang, je v souboru *ErlangGenerator.cs*.
V něm je třída ErlangGenerator, která dědí abstraktní třídu Generator.Konstruktor třídy ErlangGenerator má jako argumenty rozparsovaný původní program v e-PFL, načtenou konfiguraci systému ze souboru XML a instanci třídy StreamWiter, která slouží k tisku řetězců, což už budou kousky kódu přímo v jazyce Erlang, do finálního souboru (je vždy pojmenován jako *erlang_code.erl*). Klíčovou je funkce Generate(), ve které probíhá zápis do souboru a ze které jsou volány všechny další funkce.
Výsledný soubor se zdrojovým kódem je při tvorbě pomyslně rozdělen na několik částí, jež jsou reprezentovány pomocí třídy na práci s řetězci StringBuilder. Jak program běží, každý StringBuilder se naplňuje daty, až jsou nakonec vytištěny do finálního souboru. Ve funkci Generate() je umístěna smyčka foreach, která prochází všemi zařízeními načtenými z konfigurace, a poskytuje informace o jejich jménech, názvech a počtech vstupů a výstupů a názvech jejich vestavných procesů. Tyto informace, nezbytné pro vygenerovaný cílový kód, jsou vždy nějakým způsobem vloženy na příslušné místo do příslušného StringBuilderu.
První z těchto pomyslných částí jsou dva řádky na začátku dokumentu, v nichž je uveden název souboru (respektive modulu) a export všech funkcí, které musejí být viditelné také zvenku souboru. Jde o funkce run/0, sw_receiver/1 a o všechny funkce zodpovědné za chod zařízení (to znamená devX/0, input_box/3, tree_holder/1 a process_caller - ten má počet argumentů závislý na počtu vstupů do zařízení), které má každé zařízení svoje, a proto musí být taky všechny uvedeny na tomto řádku s exporty. Tato část kódu je vytvořena v metodě CreateExportLine, která podle počtu zařízení vloží na řádek s exporty příslušný počet funkcí. Za tyto dva řádky je vytištěn obsah souboru *ErlangIncludes.ctp*.
Následuje vytvoření řetězce odpovídajícího funkci ke spouštění systému - funkci run. Metoda CreateRunLogic projde všechna zařízení v konfiguraci, aby mohla vytvořit seznam Dev_list, ve kterém jsou názvy všech zařízení spolu s jejich identifikátory PID. Potom je do řetězce přidán řádek sloužící ke spuštění procesu přepínače a rozeslání je PID všem zařízením. Výsledná funkce run je potom metodou navrácena už přímo jako řetězec.
Později přichází na řadu tisk mechanismů všech zařízení společně s generací jejich stromo-

vých struktur s vestavnými procesy. Stromy jsou generovány v metodě CreateTree, která musí tento strom v konfiguraci projít a analogicky tuto strukturu přepisovat do takového tvaru, jaký jsem na začátku zvolil. Při průchodu stromem mohou nastat dvě možnosti - buď se jedná o fair nebo unfair uzel (to znamená, že obsah celého uzlu je uzavřen do hranatých závorek) nebo se jedná o proces, tedy o list (ten je naopak uzavřen složenými závorkami). V případě, že se jedná o list s procesem, je z konfigurace potřeba ještě získat seznam s názvy jeho vstupů, o jejichž správné vytištění se stará metoda CreateArgField. Metoda CreateArgField má dva argumenty, jimiž je struktura SortedDictionary obsahující názvy komunikačních kanálů daného zařízení a potažmo i názvy reprezentující aktuální hodnoty v nich obsažené, a *flag* určující, jestli má metoda vracet pouze názvy kanálů nebo pouze názvy proměnných, popřípadě obojí. Výsledkem metody je vždy řetězec obsahující požadovaný seznam prvků, jež bude umístěn na příslušné místo v přeloženém kódu (metoda se používá i v jiných částech programu). Strom, už jako řetězec, je potom předán metodě CreateDeviceLogic, kde je vložen jako argument vygenerované funkci tree_holder, a v níž je řádek po řádku tištěna každá funkce (jejíž název obsahuje prefix s názvem příslušného zařízení) mechanismu zařízení. Tam, kde je nutné vypsat konkrétní argumenty z konfigurace, je použita opět metoda CreateArgField, v případě generování zprávy odesílané přepínači z funkce process_caller je použita metoda CreateOutputList. Ta má velmi podobnou funkčnost jako metoda CreateArgField, s tím rozdílem, že generuje seznam s páry, obsahujícími názvy výstupů daného zařízení, a k nim korespondující názvy zařízení, pro které jsou tyto kanály vstupy. V příloze je pro ukázku metoda CreateTree, která průchodem konfiguračním souborem vytváří strom v takovém tvaru, v jakém byl předtím definován v jazyce Erlang.

Po tisku zařízení následuje překlad a tisk poslední části zdrojového kódu, což jsou vestavné procesy (funkce). O to se stará metoda GenerateFunctions. Těla funkcí s výrazy a různými konstrukcemi jsou překládány v metodě GenerateExpression. Ta postupně prochází jednotlivé výrazy obsažené v jedné e-PFL funkci, načtené z konfigurace, a přímo je překládá do ekvivalentního kódu v jazyku Erlang. Jelikož syntaxe těchto výrazů se v obou jazycích nějak výrazně neliší, většinou se jedná pouze o nahrazování znaků, které mají stejný význam. Například v e-PFL se pro zjištění prvku na nějaké pozici v seznamu používá operátoru ++, v jazyce Erlang je na tuto operaci nutné použít funkci lists:nth . A tak konstrukce pro návrat *n-tého* prveku ze seznamu *L* bude vypadat jako: L ++ n, respektive lists:nth (n, L), s tím, že hodnoty proměnných L a n jsou známé z kofigurace.

Tím je v podstatě generace cílového kódu v jazyce Erlang ukončena. Pokud bych měl shrnout to, jakým způsobem probíhala implementace třídy ErlangGenerator, uvedu, že klíčovou byla implemetace systému v programovacím jazyce Erlang doplněná o data z konfigurace. Jak jsem již řekl výše, běžné jazykové konstrukce, jako třeba funkce, se v jazycích e-PFL a Erlang moc neliší. Nejvíc práce mi dal přepis mechanismu celého vestavného systému do jazyka Erlang. Ve fázi samotného překladu už se v podstatě jednalo pouze o to, tento předem definovaný model doplnit o konkrétní data z právě překládaného návrhu systému, což v praxi znamenalo v podstatě pouze práci s řetězci.

6.4 Aktuální stav implementace

V současném stavu umí kompilátor přeložit výrazy, základní datové struktury, funkce (to znamená její deklaraci a tělo) a hlavně celý mechanismus vyjadřující systém složený z daného počtu zařízení. Mezi struktury, které prozatím nejsou kompilátorem podporovány patří:

- Lokální výrazy
- Uživatelské datové typy

Lokální výrazy (*local expressions*) by mohly být v jazyce Erlang vyjádřeny ve formě *anonymních funkcí*. Jsou to v podstatě běžné funkce, s tím rozdílem, že nejsou pojmenovány. Pro použití jako lokální výrazy se hodí, mají přístup k proměnným, které jsou známé nadřazené funkci, a navíc odpadá nutnost nějak je pojmenovávat, jak by tomu bylo v případě obyčejných funkcí. Překlad lokálních výrazů by tak mohl vypadat následovně:

```
multiply :: a Integer -> (a Integer)
multiply x = (y) where {y = x * x;}
```

Výpis 17: Funkce v jazyce e-PFL obsahující lokální výraz

```
multiply(X) -> fun(X) -> X * X end.
```

Výpis 18: Funkce v jazyce Erlang obsahující lokální výraz

Lokální výraz {y = x * x} v jazyce e-PFL by tak byl do jazyka Erlang přeložen jako anonymní funkce fun(X) -> X * X end.
Při implementaci překladu uživatelských datových typů je důležité si uvědomit, že jsou v podstatě složeny ze základních datových typů (typicky ze seznamů nebo n-tic). Bylo by proto nutné (rekurzivně) projít uživatelskou struktutru, a její konečné elementy, to znamená základní datové typy, ve správném pořadí ukládat do zvolené struktury (typicky do seznamu). Druhou možností by bylo uživatelský datový typ z e-PFL ekvivalentně převést do jazyka Erlang, pouze pomocí příslušných syntaktických úprav. Tuto možnost nám umožňuje nadstavba v jazyce Erlang, která podporuje uživatelsky definované datové typy (není ještě plně podporována). První varianta je však jistě jazyku Erlang bližší, hlavně z důvodu, že je to dynamicky typovaný jazyk, a z principu uživatelské datové typy nepodporuje.

7 Ukázka vygenerovaného cílového kódu

V této kapitole bych chtěl demonstrovat vygenerovaný cílový kód a jednotlivé jeho části porovnat s původním zdrojovým kódem v jazyce e-PFL. Z ukázek jsou vypuštěny nerelevantní části kódu.
Na začátku každého souboru v jazyce Erlang je potřeba uvést jméno tohoto souboru (modulu) a všechny funkce, které má být možné zavolat zvenku toho souboru. V případě logiky zařízení to budou všechny funkce každého zařízení, které jsou spuštěny jako procesy.

```
import "Prelude.pfl"
```

Výpis 19: Kód v jazyce e-PFL, 1. část

```
-module(erlang_code).
-export([run/0, sw_receiver/1,
dev0/0, dev0_input_box/3, dev0_tree_holder/1, dev0_process_caller/2,
dev1/0, dev1_input_box/3, dev1_tree_holder/1, dev1_process_caller/2,
dev2/0, dev2_input_box/3, dev2_tree_holder/1, dev2_process_caller/2]).
```

Výpis 20: Cílový kód v jazyce Erlang, 1. část

Potom následují pomocné funkce, které se v e-PFL nevyskutují. V cílovém kódu je využívají jednotlivá zařízení pro správnou funkčnost. Dříve uvedené pomocné metody jsou z ukázky vypuštěny.

```
%funkce search

%funkce shuffle

return_item_at(Tree, [], From_index) -> lists:nthtail(From_index, Tree);
return_item_at(Tree, Position_list , From_index) ->
return_item_at( lists:nth (element(2, lists:nth (1, Position_list )), Tree),
 lists:nthtail (1, Position_list ), element(2, lists:nth (1, Position_list ))).

increase_position_counter([H|T]) -> [{element(1, H), element(2, H) + 1}] ++ T.

sw_receiver(Dev_list) -> receive
                             List -> switch(send, List, Dev_list)
                         end,
                         sw_receiver(Dev_list).

%funkce switch

extract(name, [], Res) -> Res;
extract(name, [H|T], Res) -> extract(name, T, Res ++ [element(1, H)]);
extract(value, [], Res) -> Res;
extract(value, [H|T], Res) -> extract(value, T, Res ++ [element(2, H)]).


get_device(Dev_name, []) -> not_found;
get_device(Dev_name, [H|T]) when (element(1, H) == Dev_name) -> element(2, H);
```

```
get_device(Dev_name, [H|T]) -> get_device(Dev_name, T).

init ([], S) -> S;
init ([H|T], S) -> element(2, H) ! [init, S], init(T, S).
```

Výpis 21: Cílový kód v jazyce Erlang, pomocné funkce

Ukázkový zdrojový kód obsahuje celkem 3 zařízení, přičemž dvě jsou stejná (zařízení s názvem printer je spuštěno dvakrát). Zařízení jsou definovaná v obou jazycích následovně.

```
work :: Device
work = Fair[Process multiply,Process suma]

printer :: Device
printer = Process showB
```

Výpis 22: Kód v jazyce e-PFL, 2. část

```
%Device work

dev0() -> T_process=spawn(erlang_code, dev0_tree_holder, [[fair, {multiply, [e]}, {suma, [e]}]]),
I_process=spawn(erlang_code, dev0_input_box, [null, T_process, null]),
I_process.

dev0_input_box(Input, Tree_process, Switch) ->
receive
    [init, Sw] -> dev0_input_box([], Tree_process, Sw);
    [done] -> Caller=spawn(erlang_code, dev0_process_caller, [Switch, extract(value, Input, [])]),
        Tree_process ! [search, extract(name, Input, []), Caller],
        dev0_input_box([], Tree_process, Switch);
    [input, New_input] -> dev0_input_box(Input ++ [New_input], Tree_process, Switch)
end.

dev0_tree_holder(Tree) ->
receive
    [search, Args, Process_caller] -> search(Tree, Tree, Args, self(), Process_caller, [],
          new_level),
        dev0_tree_holder(Tree);
    [update, Updated_tree] -> dev0_tree_holder(Updated_tree)
end.

dev0_process_caller(Switch, [E]) ->
receive
    [call_process, multiply] -> Tmp=[multiply(E)],
        Switch ! output_creator([{dev1, b},{dev2, d},{dev0, e}], Tmp, []);
    [call_process, suma] -> Tmp=[suma(E)],
        Switch ! output_creator([{dev1, b},{dev2, d},{dev0, e}], Tmp, [])
end.

%Device printer

dev1() -> T_process=spawn(erlang_code, dev1_tree_holder, [[unfair, {showB, [b]}]]),
I_process=spawn(erlang_code, dev1_input_box, [null, T_process, null]),
```

```
I_process.

dev1_input_box(Input, Tree_process, Switch) ->
receive
    [ init , Sw] -> dev1_input_box([], Tree_process, Sw);
    [done] -> Caller=spawn(erlang_code, dev1_process_caller, [Switch, extract(value, Input, []) ]) ,
        Tree_process ! [search, extract(name, Input, []) , Caller ],
        dev1_input_box([], Tree_process, Switch);
    [input, New_input] -> dev1_input_box(Input ++ [New_input], Tree_process, Switch)
end.

dev1_tree_holder(Tree) ->
receive
    [search, Args, Process_caller] -> search(Tree, Tree, Args, self(), Process_caller, [],
          new_level),
        dev1_tree_holder(Tree);
    [update, Updated_tree] -> dev1_tree_holder(Updated_tree)
end.

dev1_process_caller(Switch, [B]) ->
receive
    [call_process, showB] -> Tmp=[showB(B)],
        Switch ! output_creator ([], Tmp, [])
end.

%Device printer

dev2() -> T_process=spawn(erlang_code, dev2_tree_holder, [[unfair, {showB, [d]}]]),
I_process=spawn(erlang_code, dev2_input_box, [null, T_process, null ]) ,
I_process.

dev2_input_box(Input, Tree_process, Switch) ->
receive
    [ init , Sw] -> dev2_input_box([], Tree_process, Sw);
    [done] -> Caller=spawn(erlang_code, dev2_process_caller, [Switch, extract(value, Input, []) ]) ,
        Tree_process ! [search, extract(name, Input, []) , Caller ],
        dev2_input_box([], Tree_process, Switch);
    [input, New_input] -> dev2_input_box(Input ++ [New_input], Tree_process, Switch)
end.

dev2_tree_holder(Tree) ->
receive
    [search, Args, Process_caller] -> search(Tree, Tree, Args, self(), Process_caller, [],
          new_level),
        dev2_tree_holder(Tree);
    [update, Updated_tree] -> dev2_tree_holder(Updated_tree)
end.

dev2_process_caller(Switch, [D]) ->
receive
    [call_process, showB] -> Tmp=[showB(D)],
        Switch ! output_creator ([], Tmp, [])
end.
```

```
%spuštění všech zařízení

run() -> Dev_list = [{dev0, spawn(erlang_code, dev0, [])},
                     {dev1, spawn(erlang_code, dev1, [])},
                     {dev2, spawn(erlang_code, dev2, [])}],
S=spawn(erlang_code, sw_receiver, [Dev_list]),
 init ( Dev_list , S).
```

Výpis 23: Cílový kód v jazyce Erlang, 2. část

Poslední ukázka demonstruje překlad výkonných částí systému - jednotlivých funkcí (vestavných procesů).

```
multiply :: a Integer -> (a Integer, b Integer, c Integer)
multiply x = [x + 1, x * x, x * x]

suma :: a Integer -> (a Integer, b Integer, c Integer)
suma x = [x + 1, x + x, x + x]

showB :: b Integer -> ()
showB x = writeLine (show x)
```

Výpis 24: Kód v jazyce e-PFL, 3. část

```
multiply(X) -> [X + 1, X * X, X * X].

suma(X) -> [X + 1, X + X, X + X].

showB(X) -> writeLine(X).
writeLine(X) -> io:format("~w~n",[X]).
```

Výpis 25: Cílový kód v jazyce Erlang, 3. část

8 Závěr

Ústředním motivem této bakalářské práce jsou funcionální programovací jazyky a práce s nimi. Když jsem se zhruba před rokem zamýšlel nad tématem mé budoucí bakalářské práce, chtěl jsem poznat nějakou oblast informatiky, která stojí tak trochu na jejím okraji. Rozhodl jsem se pro velmi zajímavou oblast, a tou bylo funkcionální programování. To má asi nejsilnější zázemí na akademické půdě, a myslím, že je to dáno hlavně tím, že toto paradigma je svým pojetím blízké matematice, a tak se neobjevuje jen v informatice, ale i jiných oborech. Rovněž výuka funcionálních programovacích jazyků na univerzitách není ničím neobvyklým. V praxi už jejich nasazování není tak časté, rozhodně se nedá srovnávat s dnes asi nejpoužívanějším, objektově orientovaným programováním. Asi nejznámějším komerčně využívaných funkcionálním jazykem je Erlang, vyvinutý švédskou společností Ericsson, pro nasazení v telekomunikačních systémech. Objevuje se rovněž v dnes velmi populární službě *Facebook*, kde byl použit pro implemetaci *chatu*. V praxi se také objevují i jiné funkcionální jazyky, namátkou třeba OCaml, Haskell nebo F#. Původně jsem se začal učit jazyky F# a Haskell, ale hned vzápětí jsem přešel k dalšímu funkcionálnímu programovacímu jazyku, jímž byl Erlang. Na něm se mi líbí také fakt, že je komerčně využíván, a to také ve vestavných systémech. Logickým vyústěním tedy bylo, že jsem v rámci této bakalářské práce pracoval na rozšíření již existujícího překladače, jež slouží k překladu procesně funkcionálního jazyka e-PFL, aby byl schopen generovat cílový kód právě v jazyce Erlang. V praxi to pro mě znamenalo popsat vestavný systém v jazyce Erlang podobným způsobem, jakým to dělá e-PFL. Myslím, že konečný výsledek je svou funkčností srovnatelný s původním návrhem systému v e-PFL. Překladač je tedy schopen vygenerovat cílový kód, který by mělo být, po drobných úpravách, možné nasadit na konkrétní hardware.

Pokud se dalšího vývoje týče, v první fázi by bylo nutné doplnit nyní chybějící funkčnost samotného rozšíření překladače o podporu překladu lokálních výrazů a uživatelsky definovaných datových typů. Tím pádem by byl vygenerovaný cílový kód v jazyce Erlang kompletní. Následně by se jistě dala modifikovat logika celého vestavného systému, tak jak by byla v jazyce Erlang popsána. Mám na mysli hlavně určování efektivity výsledného kódu ještě při překladu, díky čemuž by bylo možné vygenerovat více variant systému se stejnou funkčností, ale rozdílnou efektivitou. To by mohlo souviset také s přidáním další implemetace vestavného systému, přičemž použití některé z těchto variant by záviselo na typu vestavného systému, pro který by byl cílový kód generován. Obecně řečeno, celý proces, začínající u návrhu vestavného systému v jazyce e-PFL a končící u výsledného kódu, je složitý a dlouhý, proto si myslím, že možností, jak upravit či zlepšit dosavadní implemetaci, je mnoho.

Na závěr bych chtěl uvést, že tato práce pro mě byla jistě přínosem, a jsem rád, že jsem poznal funkcionální programování blíž. Samozřejmě o sobě nemohu tvrdit, že jazyk Erlang ovládám naprosto perfektně. Nejen proto bych se tedy funcionálnímu programování, a hlavně programovacímu jazyku Erlang, rád věnoval i do budoucna.

9 Reference

[1] Běhálek, Marek. *Vestavný procesně funkcionální jazyk, Autoreferát disertační práce*. Ostrava, 2009.

[2] Ericsson AB. *Erlang Reference Manual User's Guide*. Verze 5.7.4, z 23. listopadu 2009. http://erlang.org/doc/.

[3] Armstrong, Joe. *Making reliable distributed systems in the presence of software errors*, Stockholm, 2003. http://erlang.org/download/armstrong_thesis_2003.pdf.

[4] Däcker, Bjarne. *Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction*, Stockholm, 2000. http://www.erlang.se/publications/bjarnelic.pdf.

[5] Pickering, Robert. *Foundations of F#*. První vydání. Apress, 2007. ISBN 978-1-59059-757-6.

[6] Barklund, Jonas; Virding, Robert. *Erlang 4.7.3 Reference Manual*, 1999. http://erlang.org/download/erl_spec47.ps.gz.

[7] Cesarini, Francesco; Thompson, Simon. *Erlang programming: A Concurrent Approach to Software Development*. První vydání. O'Reilly Media, červen 2009. ISBN 978-0-596-51818-9.

[8] Microsoft. *.NET Framework Developer Center*. http://msdn.microsoft.com/en-us/vcsharp/default.aspx.

[9] University of St Andrews. *The Hume Programming Language*. http://www-fp.cs.st-andrews.ac.uk/hume/index.shtml.

[10] Microsoft. *.NET Micro Framework*. http://www.microsoft.com/netmf/default.mspx.

[11] Peyton Jones, S.; Gordon, A.; Finne, S. *Concurrent Haskell*. In POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA: ACM, 1996, ISBN 0-89791-769-3.

[12] The University of Warwick. *The EDEN Programming Language*. http://www2.warwick.ac.uk/fac/sci/dcs/research/em/software/eden/.

[13] Wallace, M.; Runciman, C. *Extending a functional programming system for embedded applications*. Softw. Pract. Exper., ročník 25, č.1, 1995: s.73-96, ISSN 0038-0644.

A Zdrojové kódy

```
private String CreateTree(Device dev, TreeNode treeNode, StringBuilder sbTree)
{
    bool singleProcess = true;

   if (treeNode is FairNode)
   {
    singleProcess = false;
    sbTree.Append("[fair, ");

     BranchNode bNode = treeNode as BranchNode;
      if (bNode != null)
      {
        foreach (TreeNode node in b.ChildNodes)
        {
            CreateTree(dev, node, sbTree);
        }
      }
      sbTree.Remove(sbTree.Length − 2, 2);
      sbTree.Append("]");
    }
   else if (treeNode is UnfairNode)
   {
    singleProcess = false;
    sbTree.Append("[unfair, ");

     BranchNode bNode = treeNode as BranchNode;
      if (bNode != null)
      {
        foreach (TreeNode node in bNode.ChildNodes)
        {
            CreateTree(dev, node, sbTree);
        }
      }
      sbTree.Remove(sbTree.Length − 2, 2);
      sbTree.Append("]");
    }
   else if (treeNode is ProcessNode)
   {
    ProcessNode pNode = treeNode as ProcessNode;
      sbTree.Append("{" + pNode.EmbeddedProcess.OriginalName +
      ", [" + CreateArgField(dev.Input, 2) + "]}, ");
   }

   if (singleProcess)
   {
    return "[unfair, " + sbTree.ToString().Trim().Substring(0, sbTree.Length − 2) + "]";
   }
   else { return sbTree.ToString(); }
}
```

Výpis 26: Průchod konfigurací

```
search(Tree, [H|T], Args, T_process, P_caller, Pos_list, new_level) ->
search(Tree, T, Args, T_process, P_caller, [{H, 1}] ++ Pos_list).

search(Tree, [], Args, T_process, P_caller, Pos_list) ->
search(Tree, return_item_at(Tree, lists:reverse( lists:nthtail (2, Pn_list)), element(2, lists:nth
    (2, Pos_list))), Args, T_process, P_caller, lists:nthtail (1, Pos_list));

search(Tree, [H|T], Args, T_process, P_caller, Pos_list) when (is_atom(H)) ->
search(Tree, T, Args, T_process, P_caller, increase_position_counter( Pos_list));

search(Tree, [H|T], Args, T_process, P_caller, Pos_list) when ( is_list (H)) ->
search(Tree, H, Args, T_process, P_caller, increase_position_counter( Pos_list), new_level);

search(Tree, [H|T], Args, T_process, P_caller, Pos_list) when (element(2, H) == Args) ->
P_caller ! [call_process, element(1, H)],
Updt_tree = shuffle(Tree, lists:reverse (increase_position_counter( Pos_list))),
T_process ! [update, Updt_tree];

search(Tree, [H|T], Args, T_process, P_caller, Pos_list) ->
search(Tree, T, Args, T_process, P_caller, increase_position_counter( Pos_list)).
```

Výpis 27: Funkce search

B Obsah CD

K této práci přiložené CD obsahuje:

- Textový soubor s informacemi.
- Kompletní kompilátor jako projekt spustitelný ve vývojovém prostředí Visual Studio.
- Instalační balík obsahující nástroje potřebné pro kompilaci a spuštění programů v jazyce Erlang, spolu s dokumentací. Jedná se o verzi R13B03.